MSFC-308-Paper

ShareSync: A Solution for Deterministic Data Sharing over Ethernet

Daniel J. Dunn II*, William A. Koons†, Richard D. Kennedy‡, and Philip A. Davis§
Miltec Systems, Huntsville, AL, 35806

As part of upgrading the Contact Dynamics Simulation Laboratory (CDSL) at the NASA Marshall Space Flight Center (MSFC), a simple, cost effective method was needed to communicate data among the networked simulation machines and I/O controllers used to run the facility. To fill this need and similar applicable situations, a generic protocol was developed, called ShareSync. ShareSync is a lightweight, real-time, publish-subscribe Ethernet protocol for simple and deterministic data sharing across diverse machines and operating systems. ShareSync provides a simple Application Programming Interface (API) for simulation programmers to incorporate into their code. The protocol is compatible with virtually all Ethernet-capable machines, is flexible enough to support a variety of applications, is fast enough to provide soft real-time determinism, and is a low-cost resource for distributed simulation development, deployment, and maintenance. The first design cycle iteration of ShareSync has been completed, and the protocol has undergone several testing procedures including endurance and benchmarking tests and approaches the 200µs data synchronization design goal for the CDSL.

Nomenclature

API	=	Application Programming Interface
CDSL	=	Contact Dynamics and Simulation Laboratory
COTS	=	Commercial-off-the-shelf
DLL	=	Dynamic Link Library
GUI	=	Graphical User Interface
HWIL	=	Hardware-In-The-Loop
IRQ	=	Interrupt Request
ISS	=	International Space Station
MSFC	=	Marshall Space Flight Center
NI	=	National Instruments
NIC	=	Network Interface Card
OS	=	Operating System
RTOS	=	Real Time Operating System
SCRAMNet	=	Shared Common RAM Network
TCP	=	Transmission Control Protocol
UDP	=	User Datagram Protocol
VME	=	Virtual Machine Environment

I. Introduction

As hardware-in-the-loop (HWIL) simulation laboratories begin to take advantage of today's readily-available, low-cost, distributed computing resources, reliable communication solutions are necessary to guarantee that data is available to those who need it real-time. ShareSync is a publish-subscribe communication protocol designed for reliable, low latency data replication between distributed computing nodes. The need for such a protocol became

* Systems Engineer, Space Programs Division, Member AIAA.
† Director of Programs, Space Programs Division.
‡ Software Engineer, Space Programs Division.
§ Systems Engineer, Space Programs Division.

apparent while upgrading the simulation infrastructure to the Marshall Space Flight Center (MSFC) Contact Dynamics Simulation Laboratory (CDSL).

Other solutions at both the hardware and software levels were considered. SCRAMNet (Shared Common RAM Network), for example, is a proven hardware solution that has been used in real-time applications that require low latency such as avionics and aircraft simulators[1]. It is reasonably priced for real-time hardware, well-tested, and has data distribution rates in the nanosecond range. However, reflective memory costs are concentrated at the component level; each machine, or node, attached to the simulation network must have its own proprietary reflective memory Network Interface Card (NIC). Reflective memory also has physical distance limitations, with little to no existing infrastructure, such as fiber optic cables, to connect distant facilities. The primary objective of the ShareSync protocol is to provide similar benefits utilizing non-proprietary network solutions such as Ethernet.

Ethernet is a low-cost hardware alternative, because its costs are concentrated at the infrastructure level; one switch, hub, or router can service many attached nodes, and Ethernet NICs are relatively inexpensive compared to SCRAMNet. Further, Ethernet is the *de facto* standard for generic computer networking, which allows tremendous flexibility regarding interconnectivity with other systems. Ethernet can also leverage large existing networks, including the Internet, and enjoys broad, aggressive development efforts by many standards organizations, and hardware manufacturers. The primary drawback to using Ethernet hardware for real-time simulations is its non-deterministic behavior in the event of packet collision on the wire. This problem, however, can be readily solved by using a full-duplex switched Ethernet infrastructure. Ethernet requires higher-level protocol to transfer data deterministically.

II. ShareSync Network Protocol

ShareSync is a light-weight soft real-time network protocol based on a publish-subscribe data brokerage model. Data is made available to other nodes by publishing it via the ShareSync software Application Programming Interface (API). Likewise, nodes can make requests for the data they require by declaring a corresponding subscription. The current implementation of ShareSync utilizes the User Data Protocol (UDP) as its transport layer across the network. This allows increased determinism, as opposed to Transmission Control Protocol (TCP), and also permits data packets to potentially be simultaneously broadcast to multiple IP addresses.

ShareSync is designed to facilitate soft real-time systems. *Soft* real-time generally refers to systems that have deadlines for computational frames, but are not hard-wired in such a way that it is always possible to meet the deadlines, or have limited/mild consequences for failure to complete on-time[2]. Soft real-time systems may encounter frame overruns due to software failures or over-burdening of the computational resources, but are generally more flexible and can utilize more generic, lower-cost hardware. A *deterministic* system is defined for the purpose of this paper as a repeatable, predictable system with only minor tolerable variance of latency. A real-time system with predictable and repeatable inputs and a repeatable initialization is usually considered to be deterministic.

ShareSync employs a turn-order solution based on IP address to keep the nodes communicating with each other and to keep updates consistent within a synchronization cycle. As shown in Fig. 1, each subscribing node allocates a block of memory for every other publishing node on the network. Each shared-memory block contains the variables that all nodes share with each other on the network. With this turn-order solution, the first node, which by default is the node with the lowest IP address, initiates data synchronization by broadcasting its data packets to every other node in the network. The remaining nodes are in a wait state until data from the previous node in the turn order is received. Once the data from the previous node is received, a node will then transmit its data to all other nodes and then return to the wait state to receive data from other nodes in the network. The process continues until all nodes have sent their data and received data from all of the other nodes.

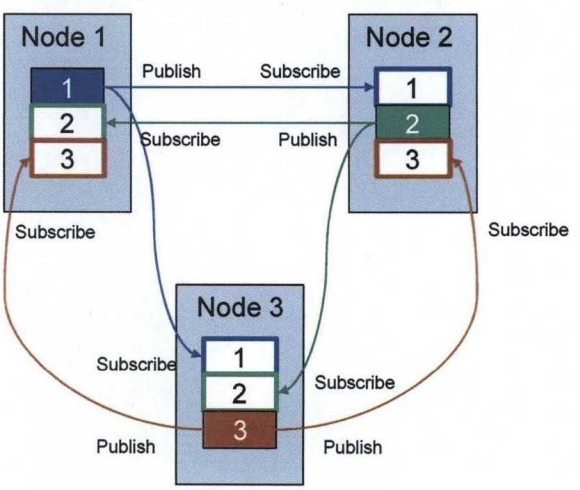


Figure 1. ShareSync Data Synchronization

ShareSync utilizes the UDP protocol to transport packets because of its reduced overhead. UDP was chosen over TCP because it provides packet delivery confirmation. This additional overhead creates scalability and determinism problems. Lost packets are re-transmitted after a random delay time, which is an unacceptable behavior for a real-time protocol. UDP sends data packets raw and does not require confirmation. Further, TCP requires an explicit connection between every pair of machines communicating on the network, incurring non-scalable overhead in computation and bandwidth. UDP can broadcast or multicast packets to every machine on the network at once. Delivery confirmation is still achieved due to ShareSync's turn-order system design. Lost packets are easily detected, logged, and the application or simulation is notified. The simulation is then responsible for handling the loss of data by terminating if appropriate.

The ShareSync API was designed to be intuitive and easy-to-use. A "ShareSync" singleton class is used to provide a namespace for all types and methods associated with the protocol, while a "Channel" class template is used to abstract the data that is to be shared between nodes. If a node needs to publish a variable, it has only to declare it using the Channel template and call the publish() method. Similarly, a subscribing node would make the exact same variable declaration, only it would call the subscribe() method instead. An example of a publishing and subscribing node is shown in Fig. 2.

```
// Publishing node
#include "ShareSync.h"
int main(int argc, char *argv[])
{
   // Declare a channel
   ShareSync::Channel<int> x = some initial value
   // Configure ShareSync
   ShareSync::configure("./sharesync.cfg");
   // Label and publish a channel
   x.publish("Variable 1");
   // Update all channels and start the real-time phase
   ShareSync::Errorcode errorCode = ShareSync::initRuntime();
   // Simulation loop
   for(;;)
   {
      x = some new value
      ...
      // Update all channels
      ShareSync::synchronize();
   }
}
```

```
// Subscribing node
#include "ShareSync.h"
int main(int argc, char *argv[])
{
   // Declare a channel
   ShareSync::Channel<int> x;
   // Configure ShareSync
   ShareSync::configure("./sharesync.cfg");
   // Subscribe to a channel with name "Variable 1".
   x.subscribe("Variable 1");
   // Update all channels and start the real-time phase
   ShareSync::Errorcode errorCode = ShareSync::initRuntime();
   // Simulation loop
   for(;;)
   {
      Use x here.
      ...
      // Update all channels
      ShareSync::synchronize();
   }
}
```

Figure 2. ShareSync Publish and Subscribe Examples

In this example a subscribing node accesses an individual piece of data on a publishing node – in this case, an integer, x. Because ShareSync is based on C++ templates, however, any data type may be shared, including user-defined structs and classes. To use ShareSync, the configure() method must first be called to configure it. Parameters are specified in a simple, XML-based configuration file. Once all channels have been declared and either published or subscribed, the calling code invokes the initRuntime() method, which initializes ShareSync with all non-local ShareSync counterparts. That is to say, initRuntime() is a blocking call that waits for a handshake signal from all other nodes using ShareSync. The program will not proceed until each node has reached the same point in execution. When all nodes are finally ready, all channels are then updated (synchronized) implicitly and the real-time portion of the simulation begins. When the end of the simulation loop is reached, it is also necessary for ShareSync to resynchronize using the synchronize() call. This function also blocks until all other nodes reach this point.

Nodes may publish or subscribe to any number of data channels. Channels are designated in ShareSync by globally persistent variable names that are specified in corresponding calls to publish() and subscribe(). If a node attempts to subscribe to a channel that has not been published, an error message is issued.

In addition to the ANSI/ISO C++ compatible version, a C-callable version of ShareSync also exists. The C version operates identically to the C++ version in terms of implementation and functionality. The C adaptation differs only in the API, in that void pointers are used in lieu of templates. A C callable version was needed due to some limitations of the National Instrument's LabVIEW software not being able to call C++ methods imported through a Dynamic Link Library (DLL). Full compatibility with the LabVIEW software was desired, since National Instruments is a leader in measurement and automation, and their products are often used in HWIL simulation laboratories, including the CDSL.

III. Contact Dynamics Simulation Laboratory

The initial avionics lab where ShareSync will be tested and deployed, the CDSL, provides NASA with unique, world-class capabilities to perform real-time HWIL simulation and testing of docking and berthing mechanisms and associated control systems. Simulation support provided by the CDSL includes the testing and validation of the ISS Common Berthing Mechanism, Space Shuttle Remote Manipulator System End Effector, and the Orbital Maneuvering Vehicle Three Point Docking Mechanism[3]. The laboratory test stand consists of a Six Degree of Freedom (6DOF) platform, shown in Fig. 3, and controlled by six hydraulic legs. With computerized motion transforms, these six legs allow the platform to move in the six Cartesian degrees of freedom: x, y, z, roll, pitch, yaw. Directly above the 6DOF platform is a ceiling mounted stationary platform with limit switches, mechanical isolation, and force / moment sensors installed, both to measure the encounter forces and to protect the test article and building structure should a collision become too forceful. A block diagram showing a more detailed breakdown of the simulation model is shown in Fig 4. The equipment currently used to run the simulation consists of an SGI Challenge 10000 XL which runs the all the simulation code and a Virtual Machine Environment (VME) I/O system. The VME system interfaces the simulation code to the

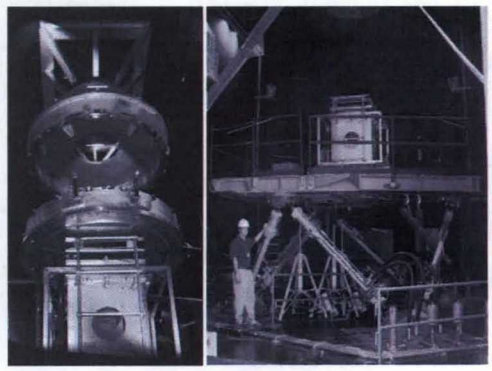


Figure 3. CDSL Test Stand[3]

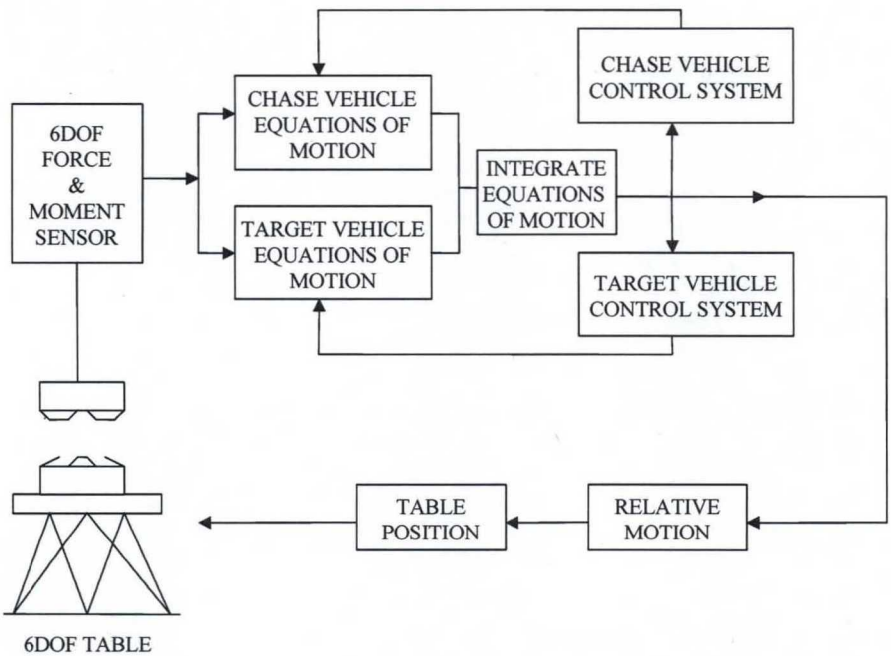


Figure 4. Two Body Docking Model Simulation[3]

sensors and hydraulics of the test stand, and additional scenario-specific sensors provided by the lab's customer.

The main reason for upgrading the CDSL is to move towards a more common simulation environment within the NASA centers and research facilities. This common simulation environment utilizes the Trick[4] Simulation Toolkit which was written for NASA and is used extensively at the Johnson Space Center (JSC) in both real-time and non real-time simulations. The Trick Simulation Toolkit operates in UNIX and Linux Operating Systems (OS) and can utilize standard commercial-off-the-shelf (COTS) PCs, which are more powerful and cost much less than the older SGI Challenge cost to maintain. A block diagram of the new CDSL infrastructure is shown in Fig. 5.

The new infrastructure consists of three 1 Gb Ethernet LANs, all of which are isolated from each other in order to control and optimize bus utilization. The developer bus is intended to handle asynchronous data communications between the simulation control graphical user interface (GUI) and the simulation nodes and the I/O node. The Trick

Simulation Toolkit and models are installed and configured on the simulation nodes, which will utilize the simulation bus for real-time Trick distributed simulation communication. The I/O node, responsible for interfacing with the facility hardware, will utilize ShareSync on the private I/O bus network for real-time data synchronization between the Trick simulation models and the I/O node.

The simulation nodes are high-end COTS PC workstations with two AMD Opteron dual-core processors, 8 GB of system memory, and the Fedora Core 5 Linux OS. While not a Real Time Operating System (RTOS), with Fedora Core 5, background processes and interrupts can be isolated to specific processors. Additionally, the Trick Simulation Toolkit has been tested to work with this particular Linux distribution.

The I/O node is a National Instruments (NI) PXI-1045 chassis running LabVIEW Real-time (RT) v8.2 on the PharLap ETS version 12.0 OS with a 2 GHz processor and 2 GB of RAM. NI's LabVIEW RT programming environment was chosen for the I/O node due to the abundance of drivers already written supporting both NI's data acquisition and interface cards, as well as, PXI and cPCI cards from other companies.

IV. Implementation and Test

A beta version of the ShareSync protocol was developed according to the design specification, and verification and validation testing was performed at the MSFC CDSL. The protocol was compiled for Microsoft Windows and Linux OSs. The primary test objectives were to verify the software implementation, to demonstrate the protocol's data sharing capabilities, and to test the protocol's reliability using UDP on the dedicated Gb Ethernet I/O LAN.

The first test scenario utilized a simple test driver to interface with the ShareSync Protocol API to synchronize data packets of varying sizes between two Linux workstations. Following successful testing between two workstations, additional nodes were added to the test configuration. Whereas the Linux workstations were representative of the CDSL simulation nodes, these additional nodes added to the test configuration consisted of Dell Inspiron laptops with 100 Mb LAN ports and running Microsoft Windows XP. The primary purpose of adding these nodes was to validate the flexibility of the protocol to synchronize data on differing platforms. The test configuration is shown in Fig. 6.

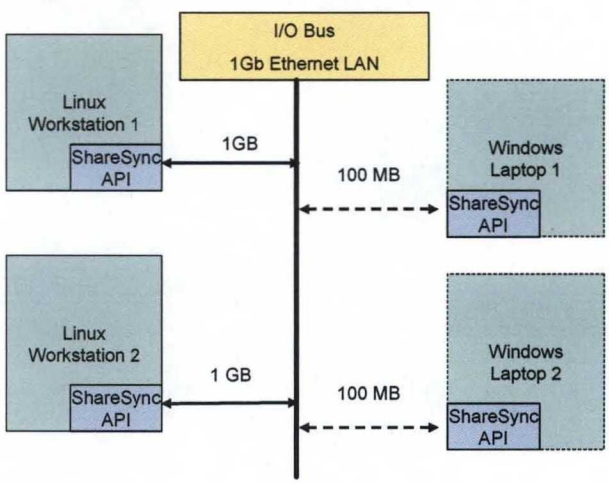


Figure 6. ShareSync Test Configuration

The initial test with the two Linux workstations focused on increasing the size of the data packet from a single byte up to 64 kB, the maximum size packet that can be communicated using UDP without splitting data into multiple packets. After verifying that ShareSync successfully synchronized packets of various sizes, the next step in the testing process involved executing the protocol for an extended period of time to verify that the ShareSync protocol using UDP would be able to support a simulation without losing packets. ShareSync is designed to timeout after a user-specified duration should a loss of data occur; however, with a dedicated full-duplex Ethernet switch, it is anticipated that the occurrence of lost data will be rare. A test driver was developed to publish and subscribe 4 bytes of data on each node as fast as possible for an indefinite period of time. This test driver was run on several occasions for up to 15 hours per test, and no packets were dropped in any of the test cases. For the purpose of the CDSL application, these results are encouraging because a typical CDSL simulation usually lasts on the order of a few minutes.

The final phase of the verification process involved adding the additional laptop nodes to the data sharing network. Simulations using this configuration were also run for extended periods of time, exceeding 10 hours in duration, without experiencing any packet loss. While benchmarking was not attempted during any of the verification and validation testing activities, results of the exercise proved that the ShareSync protocol implementation will perform reliably within the context of the CDSL and is flexible enough to support multiple nodes of varying platforms.

Following the successful verification and validation testing activities, ShareSync was integrated with the Trick Simulation Toolkit and NI's LabVIEW™ Real-time v8.2 programming language and the laboratory infrastructure was configured as shown in Fig. 5. While the CDSL Trick simulation models were not integrated at this point, the

objective was to integrate the protocol into the actual HWIL infrastructure and benchmark the performance of ShareSync when communicating between a simulation node and the PXI I/O chassis.

The CDSL hardware facility requires a minimum simulation cycle of 2 milliseconds per frame as illustrated in Fig. 7. Minimally, the new infrastructure, must meet this requirement. While simulation performance is expected to increase significantly, the design goal is to allocate 90% of the 2 millisecond frame to the simulation execution and the remaining 10% to data synchronization. The expected size of the data that will be shared between the simulation and I/O nodes is approximately 800 bytes. Thus ShareSync must be able to complete data synchronization of 800 bytes within 200μs. The objective of the benchmarking activity was to verify that ShareSync can deterministically meet this design goal.

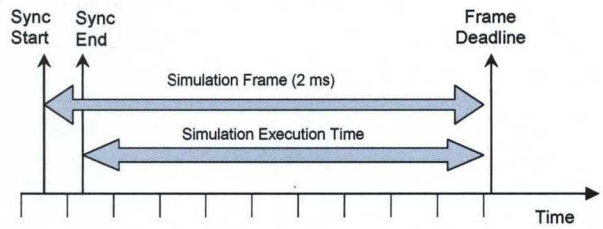


Figure 7. CDSL Simulation Frame

After developing test drivers in Trick and LabVIEW, test cases were developed to benchmark the synchronization cycle performance for data packets ranging from 100 bytes to 2,000 bytes. For benchmarking purposes, the I/O node measured the time difference between synchronization from one frame to the next and stored the results into an array. Using the Linux *isolcpus* command all of the background process running in the OS were isolated to processor 0 of the four processor system. Additionally, the Linux command *noirqbalance* was used to assign all of the interrupt request (IRQ) activities to processor 0. With the processors isolated, the IRQ activity for the interrupt containing the Ethernet device was assigned to processor 2 and the simulations were executed so that they would only run on processor 3, effectively isolating the simulation and the Ethernet bus from the other processes running on the simulation node.

The simulation node was configured with a test driver built in the Trick environment that was set up to synchronize every 2 milliseconds, essentially creating a 2 millisecond frame. Tests were conducted to profile the synchronization latency for data packet sizes between 100 and 8,000 bytes. Each test case consisted of 30,000 synchronization events several iterations were executed. For the test, 1Gb PCI NICs were used to connect the simulation node and I/O node to the 1 Gb Ethernet switch The results of these tests produced average synchronization times that did not vary much at all up to 2,000 bytes as shown in Fig. 8.

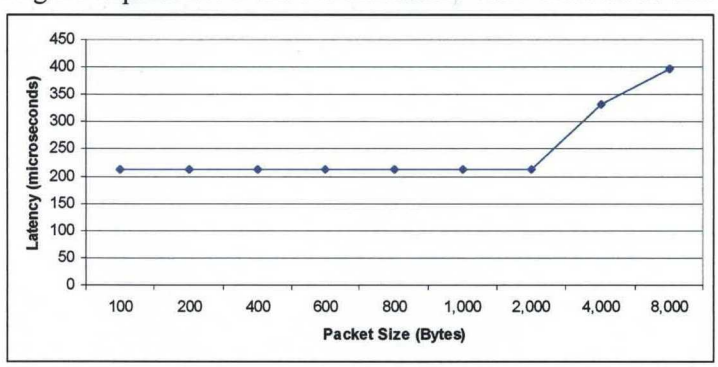
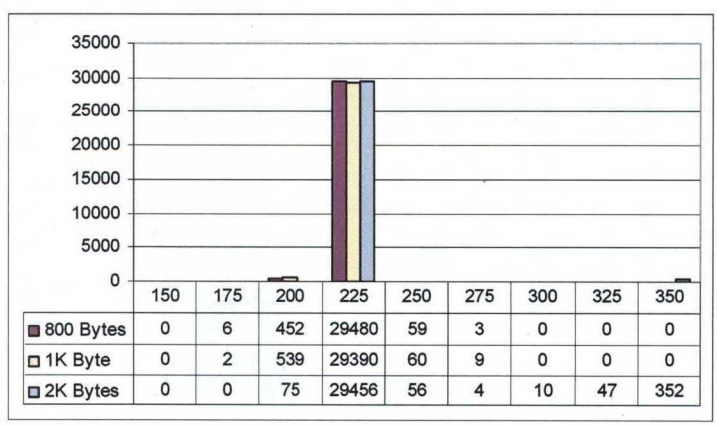


Figure 8. ShareSync Latency

Fig. 9 illustrates the determinism of ShareSync in the tested environment. Over 98% of the synchronization delays for the 800 byte, 1,000 byte, and 2,000 byte test scenarios were within 25μs of each other. These results are typical for all of the test cases that were performed using the PCI LAN port. While the average synchronization latency exceeded the 200μs design goal, the overall results of the testing activities are encouraging and indicate that the ShareSync protocol is capable of providing a soft real-time deterministic communication option for the CDSL.



	150	175	200	225	250	275	300	325	350
800 Bytes	0	6	452	29480	59	3	0	0	0
1K Byte	0	2	539	29390	60	9	0	0	0
2K Bytes	0	0	75	29456	56	4	10	47	352

Figure 9. ShareSync Synchronization Latency Histograms

V. Conclusion

The ShareSync protocol has undergone its first phase of testing and has been shown to be a reliable means of communication, capable of meeting the real-time goals and requirements of the CDSL, using only COTS computer hardware. The results from the first phase of testing showed that ShareSync can perform for an extended duration, well beyond the normal operation duration for a simulation using the new hardware in the CDSL. The benchmarking test results revealed that ShareSync executing on the Gb Ethernet I/O node bus is capable of supporting deterministic data sharing in the 200 μs – 225μs range. Additional tests were conducted utilizing the simulation node's onboard Gb Ethernet port which easily meet the 200μs average goal, however, in this configuration, the system suffers from occasional non-deterministic outliers (overruns) in the 10 millisecond range, which is believed to be caused by the Linux drivers installed. Further testing is required, however, these results suggests that improved hardware, such as PCI Express (PCI-E) NICs, 10 GbE switches and NICs, enhanced onboard drivers, and potential optimizations to the software will improve the protocol's performance.

The next step is to incorporate ShareSync into a fully developed simulation with Trick and LabVIEW™, first running a mock-up of the CDSL's test platform and eventually running the actual test platform. These tests would allow the actual integrated cycle times to be determined. The benchmark results, while above 200μs at this time, are still well within acceptable operational parameters for the CDSL. Further tests will determine the impact of additional nodes, more data, and possibly other base protocols on synchronization cycle time.

Acknowledgments

The authors would like to acknowledge the support of the MSFC Avionics System Test Branch – Ms. Linda Brewster, Mr. Drew Hall, and Mr. Nick Johnson, and our Jacobs Team Lead Mr. Bobo Hand.

References

[1] T. Bohman, "Shared-Memory Computing Architectures for Real-Time Simulation-Simplicity and Elegance", Technical report, Systran Corporation, 1994.

[2] P Doyle, "Introduction to Real-Time Ethernet I", *The Extension: A Technical Supplement to Control Network* [online journal], Vol. 5, Issue 3, May-June 2004, URL: http://www.datalinkcom.net/Real Time Ethernet1.pdf [cited 15 April 2007].

[3] D. Hall, B.M. Sloane, P. Tobbe, "Modeling and testing of docking and berthing mechanisms", Technical Whitepaper, Aeronautics and Space Administration, Marshall Space Flight Center, AL, USA 35812; Dynamic Concepts, Inc., 6700 Odyssey Drive, Suite 202, Huntsville, AL, USA 35812

[4] E. Paddock, A. Lin, K. Vetter, E. Crues, "TRICK: A Simulation Development Toolkit", AIAA Modeling and Simulation Technologies Conference & Exhibit, August 2003, AIAA 2003-5809.